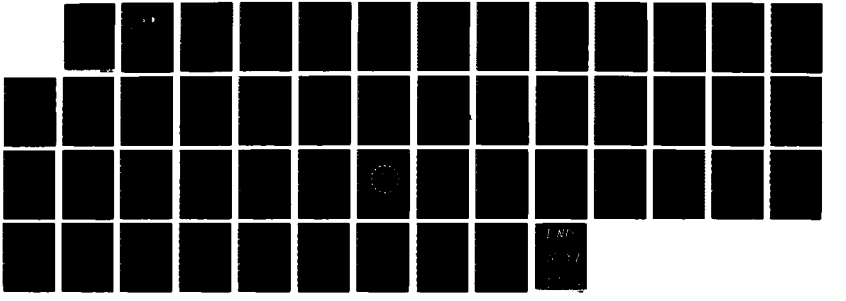
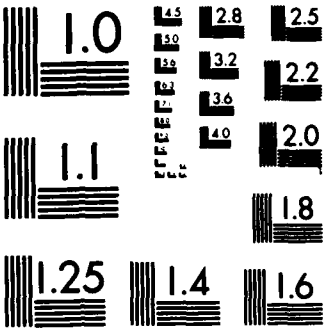
AD-A184 792 TOKEN EXECUTION STRATEGIES FOR DISTRIBUTED ALGORITHMS: SIMULATION STUDIES(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB M J LLOYD JUL 87 1/1

UNCLASSIFIED UILU-ENG-87-2245 N00014-85-K-0570 F/G 12/7 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

July 1987

UILU-ENG-87-2245
ACT-81

COORDINATED SCIENCE LABORATORY
College of Engineering
Applied Computation Theory

TOKEN EXECUTION STRATEGIES FOR DISTRIBUTED ALGORITHMS: SIMULATION STUDIES

Mark James Lloyd

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

ADA184792

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ACT-#81 UILU-ENG-87-2245	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Office of Naval Research 8b. OFFICE SYMBOL (If applicable) N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-85-K- 0570
8c. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Token execution strategies for distributed algorithms: simulation studies

12. PERSONAL AUTHOR(S)
Lloyd, Mark J.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	July 1987	47

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	distributed algorithms, minimum spanning tree, token, congestion control

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Six variations of the token execution strategy of a distributed algorithm are described, applied to an algorithm for computing the minimum-weight spanning tree simulated on three network topologies, and compared with the chaotic execution strategy. In a chaotic execution, a processor may transmit a message M as soon as it generates M. In a token execution, a processor may transmit a message only when it holds a unique token. The token execution limits the number of messages in transit at any time. Execution with one token allows the user to observe the response to each message sequentially; execution with a fixed number of tokens provides a congestion control strategy. The best combination of variations of the token execution strategy uses 5.6% to 15.8% more messages and 0% to 12.3% more execution time than the chaotic execution, depending on network topology.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

TOKEN EXECUTION STRATEGIES FOR DISTRIBUTED ALGORITHMS: SIMULATION STUDIES

BY

MARK JAMES LLOYD

B.S., Iowa State University of Science and Technology, 1986

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1987

Urbana, Illinois

ABSTRACT

Six variations of the token execution strategy of a distributed algorithm are described, applied to an algorithm for computing the minimum-weight spanning tree, simulated on three network topologies, and compared with the chaotic execution strategy. In a *chaotic execution*, a processor may transmit a message M as soon as it generates M. In a *token execution*, a processor may transmit a message only when it holds a unique token. The token execution limits the number of messages in transit at any time. Execution with one token allows the user to observe the response to each message sequentially; execution with a fixed number of tokens provides a congestion control strategy. The best combination of variations of the token execution strategy uses 5.6% to 15.8% more messages and 0% to 12.3% more execution time than the chaotic execution, depending on network topology.

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Professor Michael C. Loui, for his guidance and suggestions. I would also like to thank AT&T Bell Laboratories, in particular, my supervisor Yern Yeh, for their sponsorship of my graduate education under their One Year On Campus Graduate Degree Program.

The computer resources required for this work were supported by the Office of Naval Research under Contract N00014-85-K-00570.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

A distributed system is a set of asynchronous processors connected by a set of bidirectional links. In a distributed algorithm execution, each processor executes the same local algorithm, which consists of waiting for incoming messages, changing processor state, and sending messages.

In a *chaotic execution* of a distributed algorithm, a processor may send a message M when it generates M. Several messages may be in transit at the same time.

In a *token execution* of a distributed algorithm, one or more tokens are inserted into the network at the beginning of the algorithm execution. The number of tokens remains constant throughout the algorithm execution. A processor may send a message only when it holds one of the tokens. In the Simple Token Execution Strategy presented in this thesis, a maximum of two messages is in transit for each token at any time.

The token execution strategy is interesting for two reasons. First, token execution with one token allows the user to observe the algorithm execution in a sequential manner, i.e., the response to each message can be observed sequentially. Second, since the token execution restricts the number of messages an algorithm may have in transit at any time, it can be used as a congestion control strategy.

In this thesis, the Simple Token Execution Strategy and its six variations are described. The number of messages passed and algorithm execution time due to the Simple Token Execution Strategy, its variations, and combinations of variations are studied. It is shown that for the minimum-weight spanning tree algorithm of Gallager *et al.* (1983), the best combination of variations uses 6.5% to 15.8% more messages and 0.9% to 12.3% more execution time than the chaotic execution, depending on the network topology.

2. COMPUTATIONAL MODEL

A general connected network, which is assumed in this thesis, can be modeled by a connected undirected graph in which graph nodes correspond to processors and graph edges correspond to bidirectional communication links. Each processor has a unique identifier; each link has a unique identifier. A processor does not know the identifier of the processor at the other end of each adjacent link. Processors view links from the session layer of the Open Systems Interconnection Model (Tanenbaum, 1981), i.e., messages sent on a link arrive at the other end of that link after an unpredictable but finite delay without error and in sequence.

Each processor performs the same local algorithm asynchronously with respect to other processors. The local algorithm consists of waiting for incoming messages, processing, and sending messages on adjacent links. Processing time is assumed negligible when compared with message passing time. A message consists of the originating processor's identifier, destination link identifier, message type, and appropriate arguments.

Each processor has some input data. Processors cooperate to compute some function of that data. For example, in the case of the minimum-weight spanning tree algorithm of Gallager *et al.* (1983), each processor has the weights of its adjacent links as an input value. Processors determine the links that together constitute a minimum-weight spanning tree of the network.

There are two natural conditions that could define the termination of a distributed algorithm.

(1) All processors in the network have no messages to send.

(2) All processors are in a designated terminated state.

In this thesis, the first termination condition is assumed. If the second termination condition is desired, then it is the responsibility of the distributed algorithm to put the

network processors in a designated terminated state. Processors in a designated terminated state are assumed to have no messages to send.

Algorithm executions are studied by observing the number of messages sent and the algorithm execution time. Although the algorithm must work for unpredictable but finite delays. the usual algorithm execution time measure is based on unit link delays. In this thesis. the algorithm execution time measure is based on bounded random link delays. Finite delays are bounded because of simulator memory limitations.

3. LITERATURE SURVEY

The literature survey introduces the reader to the kinds of problems solved by distributed algorithms.

One of the simplest and most studied distributed algorithms is election of a leader. Below, three election algorithms are described, each assuming a different network topology. This is followed by a description of two other algorithms, one for determining the global state of the network and the other one for simulating a synchronous network on an asynchronous network.

3.1. Election Algorithms

An election algorithm selects one processor as leader such that all processors in the network know who the leader is. The election algorithm has many applications, including selecting a processor to start a new token in a token ring and identifying the transaction coordinator in a distributed database.

For the three election algorithms described below, the computational model of Section 2 is assumed except for changes explicitly stated.

3.1.1. Election on a Unidirectional Ring

A unidirectional ring is assumed in the election algorithm presented in Peterson (1982). Processors communicate with message passing in one direction. Each processor has a unique input value, which may be transmitted and compared. The election problem is to select the processor with the maximum (or minimum) input value.

The algorithm works as follows. All processors are initially in an active state. As the algorithm progresses, some processors move to a relay state. Relay processors just pass on any messages they receive. Active processors operate in phases. In each phase, the number

of active processors decreases by at least half. All but one processor moves to a relay state by the end of the algorithm execution; the remaining active processor elects itself as leader.

Peterson's algorithm requires at most $O\ (N\ \log_2 N)$ message transmissions and $O\ (N)$ time on a ring with N processors.

3.1.2. Election in a Complete Network

A complete network is assumed in the election algorithm in Afek and Gafni (1984). Every processor has a link to every other processor in the network. Each processor has a unique input value, which may be transmitted and compared. This election problem is to distinguish one processor from all others, not necessarily the processor with the maximum (or minimum) input value.

The algorithm works as follows. Processors can be in one of three states: father, uncaptured, or captured. A captured processor is not allowed to become a father. At least one processor is initially a father; all other processors are uncaptured. As the algorithm progresses, fathers capture uncaptured neighboring processors. A father may try to recapture a captured neighboring processor by contesting that processor's father. The father who has captured the most processors wins the contest. The losing father and its captured processors become uncaptured processors. The father who succeeds in capturing all its neighbors elects itself as leader of the network.

This algorithm requires at most $2\ N\ \log_2 N$ messages and $O\ (N)$ time on a network with N processors.

3.1.3. Election in a General Network

A general network is assumed in the algorithm of Gallager *et al.* (1983). Each link has a unique weight. (When link weights are not unique, one simply appends to the link weight the unique identifiers of the two processors joined by the link.) Each processor

initially knows the weight of links adjacent to that processor.

The algorithm of Gallager *et al.* (1983) actually determines the minimum-weight spanning tree (MST) of the general network (Awerbuch, 1987). This MST algorithm constructs a tree connecting every processor rooted at a link called the network MST *core*. To use the MST algorithm for election, one of the two processors adjacent to the *core* is selected as leader of the network. Besides election, the minimum-weight spanning tree can be used as a minimum-cost broadcast route when each link weight represents the cost of message passing across that link.

This algorithm uses *MST fragments* as building blocks to build the network MST. An MST fragment is a set of connected processors together with the links that define their minimum-weight spanning tree. Every fragment computed by the algorithm is part of the MST of the entire network. As the algorithm progresses, MST fragments grow. MST fragments can grow by *absorbing* other MST fragments or *combining* with other MST fragments. Each MST fragment F_i has a value LEVEL(F_i) that reflects how many times the MST fragment has *combined* with other MST fragments. Initially, each MST fragment F_i comprises just one processor, and LEVEL(F_i) = 0.

The algorithm works as follows. Each single processor MST fragment finds its minimum-weight adjacent link. A message is sent on the minimum-weight link asking the MST fragment at the other end of the link to combine.

If two MST fragments, F_i and F_j, such that LEVEL(F_i) = LEVEL(F_j), ask each other to combine over the same link, then they *combine* into a new MST fragment, F_k, comprising F_i, F_j, and the link between F_i and F_j. The link between F_i and F_j is called the *core* of F_k. MST fragment F_k is assigned LEVEL(F_k) = 1 + LEVEL(F_i).

If a MST fragment F_i asks to combine with a fragment F_j such that LEVEL(F_i) < LEVEL(F_j), then F_j *absorbs* F_i forming a new MST fragment F_k. The new MST fragment F_k comprises F_i, F_j, and the link between F_i and F_j. MST fragment F_k is assigned

$LEVEL(F_k) = LEVEL(F_j)$.

If a fragment F_i asks to combine with a fragment F_j such that $LEVEL(F_i) > LEVEL(F_j)$, then F_i must wait until F_j asks F_i to combine on the same link, at which time, F_i absorbs or combines with F_j.

Each time a new MST fragment is formed, the processors of the new MST fragment cooperate in finding the minimum-weight link outgoing from the new MST fragment. The processor adjacent to this link sends a message asking the MST fragment at the other end of the link to combine.

The MST algorithm terminates when one MST fragment contains all network processors. The common minimum-weight link of the last two MST fragments that combine is called the *network MST core*.

This MST algorithm requires at most $2L + 5N \log_2 N$ messages on a network with N processors and L links.

3.2. Other Distributed Algorithms

Below is a description of an algorithm that determines the global state of the network and a description of an algorithm that simulates a synchronous network on an asynchronous network.

3.2.1. Determining Global State

The global state of a network is the set of processor states and messages in transit. Some examples of global state are *system is deadlocked* and *computation has terminated* and *all tokens in a token ring have disappeared*. Chandy and Lamport (1985) present an algorithm for determining the global state of the network.

The algorithm works as follows. A processor can record its own state and the state of its adjacent links; *it can record nothing else*. The algorithm begins execution from a single

processor P_1. Processor P_1 records its processor state and sends a marker message on each adjacent link. Suppose a processor P_i receives a marker on adjacent link L_j. If P_i has not yet recorded its processor state, then P_i records its processor state, records the link state of L_j as an empty sequence of messages, and sends a marker message on each adjacent link, including L_j. If processor P_i has recorded its processor state, then P_i records the link state of L_j to be all messages received on L_j since P_i recorded its processor state. When all processor and link states are recorded, they are collected to form the global state of the network.

3.2.2. Network Synchronization

In a *synchronous* network, messages are allowed to be sent only at integer times, or *pulses*, of a global clock. For each link, at most one message can be sent in each direction during each pulse. The delay of each link is assumed to be at most one time unit of the global clock. A synchronous network is simulated on an asynchronous network in the algorithm in Awerbuch (1985).

Consider a distributed algorithm A capable of being executed on the network defined by the computational model in Section 2.

Awerbuch's algorithm works as follows. Each processor executes a process called a *synchronizer* along with the local copy of algorithm A. The *synchronizer* simulates a global clock by generating clock *pulses*. A clock pulse is an indication to the local algorithm A that it can send 0 or 1 messages on each adjacent link. Processor P_i's *synchronizer* generates a clock pulse when P_i discovers that all processors in the network have received acknowledgments for all messages sent in the last time unit.

Many other algorithms have been proposed as solutions to other problems in distributed computing systems. A fairly complete review of the problems and algorithms appears in Liestman and Peters (1986).

4. TOKEN EXECUTION STRATEGY

A token execution of a distributed algorithm uses two types of messages: ordinary messages and token messages. An ordinary message is generated by a processor in accordance with the distributed algorithm it is executing. A token message is generated by a processor to transfer possession of the token to a neighboring processor.

In the Simple Token Execution Strategy and its variations, a processor may send an ordinary message only when it holds the token. It follows that ordinary messages generated by a processor P_i must be stored in an outgoing message queue $MSG(P_i)$ until P_i receives the token and is able to send ordinary messages.

Suppose processor P_i holds the token. The Simple Token Execution Strategy and its variations determine which ordinary message M from $MSG(P_i)$ is sent on M's destination link and which adjacent link L is selected to send the token message on.

In the following description of the Simple Token Execution Strategy and its six variations, assume only one token is inserted into the network at the beginning of the algorithm execution except where explicitly stated.

4.1. Simple Token Execution Strategy

The Simple Token Execution Strategy uses one token without message and time-saving heuristics. Tiwari and Loui (1986) prove that if the chaotic execution of the algorithm terminates, then the Simple Token Execution terminates with at most three times the number of messages sent by the chaotic execution.

4.1.1. Starting Token Execution

Assume processor P_1 initially holds the token and begins the algorithm execution by generating ordinary messages and enqueuing them in $MSG(P_1)$. Also, assume other

processors P_i, $i \neq 1$, are initially asleep with $MSG(P_i)$ empty. These processors are awakened by the arrival of a message, at which time they participate in the execution of the distributed algorithm.

Since P_1 holds the token, P_1 dequeues the ordinary message M from the front of $MSG(P_1)$ and sends it on the M's destination link L. P_1 now sends a token message on link L with status *Need Token Back*. The status *Need Token Back* (as opposed to *Don't Need Token Back*) indicates that P_1 ultimately needs the token back to send other ordinary messages in $MSG(P_1)$.

4.1.2. Joining the Return Path

Assume the processor P_i receives the ordinary message M_{in} and the token message sent by P_1 on link L_{in}. P_i generates ordinary messages in response to M_{in} and enqueues them in $MSG(P_i)$. P_i assigns the identifier of L_{in} to the local variable $RET(P_i)$. Processor P_i dequeues the ordinary message M_{out} at the front of $MSG(P_i)$ and sends it on M's destination link L_{out}. P_i then sends a token message with status *Need Token Back* on link L_{out}.

In general, the values of the set of RET variables from all processors define a sequence of the processors $(P_1 \ldots P_i \ldots P_k)$. Processor P_k holds the token, and for any two consecutive P_a and P_b in the sequence, P_a sent P_b the token with status *Need Token Back*. The sequence of processors is called the *Return Path* of the token. Initially, each processor P_i assigns 0 to $RET(P_i)$, indicating that P_i is not yet part of the Return Path. When P_1 begins execution, it assigns * to $RET(P_1)$, indicating that P_1 is the beginning of the Return Path. $RET(P_i)$ tells P_i which link to send the token on when P_i is finished with the token. The following property is proved in Tiwari and Loui (1986):

> *The Property of the Return Path*: If processor P_i is not on the Return Path, the $MSG(P_i)$ is empty. If processor P_i is on the Return Path, then P_i will eventually receive the token.

Suppose at a later time, P_i receives an ordinary message M_{in} on link L_{in} followed by a token message with status *Need Token Back*. P_i generates ordinary messages in response to M_{in} and enqueues them in $MSG(P_i)$. If P_i is still on the Return Path, then P_i rejects the token message, i.e., P_i sends a token message with the status *Don't Need Token Back* on link L_{in}. The token message is rejected because P_i is already on the Return Path. *The Property of the Return Path* guarantees that P_i will eventually receive the token.

4.1.3. Exiting the Return Path and Termination

Suppose P_i, with $RET(P_i) = L$, receives a token message with status *Don't Need Token Back*. This status indicates that P_i is the current end of the Return Path. If P_i's message queue is empty, then P_i exits the Return Path by sending a token message with status *Don't Need Token Back* on link L and assigning 0 to $RET(P_i)$.

Suppose P_1 receives a token message with status *Don't Need Token Back*. This status indicates that P_1 is the current end of the Return Path. $RET(P_1) = *$ indicates that P_1 is the beginning of the Return Path. In other words, P_1 is the only processor on the Return Path. *The Property of the Return Path* guarantees that all other processors P_i have empty message queues $MSG(P_i)$. If P_1's message queue $MSG(P_1)$ is empty, then the token execution of the algorithm terminates.

4.2. Six Variations of the Simple Token Execution Strategy

The six variations described below are heuristics with the goal of reducing the number of transmitted messages and length of token algorithm execution. The variations are Piggyback Token Messages, Deny Token Start Processor, Leave the Return Path, Send Multiple Messages with Each Token, Send Messages on Token Reject, and Multiple Tokens.

4.2.1. Piggyback Token Messages

In this variation, the token message and ordinary message are combined into one message whenever possible. Tiwari and Loui (1986) show that if the chaotic execution of the algorithm terminates, then the Simple Token Execution with piggybacking terminates with at most twice the number of messages sent by the chaotic execution.

4.2.2. Deny Token Start Processor

Suppose processor P_i knows which, if any, neighboring processor initially held the token. When P_i dequeues and sends the ordinary message M at the front of $MSG(P_i)$ to P_j, P_i then sends a token message to P_j if and only if P_i knows that P_j did not initially hold a token.

This strategy is valid because if P_j initially held the token, then P_j is on the Return Path. *The Property of the Return Path* guarantees that P_j will eventually receive the token.

P_i can use one of two methods to determine which neighboring processor started with the token. (Keep in mind that according to the computation model, P_i does not know the identifier of the processors on the other end of adjacent links.) In the first method, the processor that initially holds the token sends a special message on each adjacent link telling neighboring processors it initially held the token. In the second method, a processor discovers which neighbor initially held the token during the execution of the algorithm. Every token message carries a status telling whether the token message originator initially held the token. The second method is used in this thesis.

4.2.3. Leave Return Path

Suppose P_1 holds the token, and suppose P_1's message queue $MSG(P_1)$ contains exactly one ordinary message M with destination P_2. P_1 dequeues M and sends M to P_2. P_1 assigns 0 to $RET(P_1)$ and sends a token message with status *Don't Need Token Back* and

an additional flag telling P_2 to assign * to RET(P_i). P_2 becomes the beginning of the Return Path. P_1 is no longer on the Return Path. Later in the computation, P_1 could receive a token message and become part of the Return Path.

This variation saves token messages and terminates shortly after the transmission of the last ordinary message.

4.2.4. Send Multiple Messages with Each Token

Suppose P_i holds the token. P_i sends all ordinary messages in MSG(P_i) that are to be sent on the same link L as the ordinary message at the front of MSG(P_i). P_i now sends a token message with status *Need Token Back* on link L.

4.2.5. Send Message on Token Reject

Suppose P_i is already on the Return Path and P_i receives a token message with status *Need Token Back* on link L. In the Simple Token Execution Strategy, P_i *rejects* the token, i.e., P_i sends a token message with status *Don't Need Token Back* on link L.

In the Send Message on Token Reject Strategy, P_i sends the first ordinary message M in MSG(P_i) to be sent on link L. P_i now sends a token message on link L.

4.2.6. Multiple Tokens

Suppose several tokens are used in the token execution of the algorithm. A processor may send an ordinary message M only when it holds one of the tokens. Initially each processor holds at most one token. Each token carries the identifier ID_j of the processor P_j that initially held the token. A separate Return Path is maintained for each token. Each processor P_i maintains an array of N return link variables RET(P_i, ID_j), where $1 \leq j \leq N$ and N is the number of processors in the network. A processor handles token messages in the same manner as the Simple Token Execution Strategy, except now the

processor must use the Return Link corresponding to identifier of the token.

The algorithm execution time with multiple tokens is less than the algorithm execution time without multiple tokens since several messages can be in transit simultaneously.

4.3. Combinations of Variations

All six variations above can be combined except *Deny Token Start Processor* and *Leave Return Path*. To understand why Deny Token Start Processor and Leave Return Path cannot be used simultaneously, consider the following scenario. Suppose the beginning of the Return Path moves from processor P_j according to the Leave Return Path strategy. P_i sends an ordinary message to P_j but does not send a token message to P_j because P_i knows P_j initially held a token. P_j generates and enqueues ordinary messages in response to the ordinary message received from P_i. P_j is not on the Return Path and does not have an empty message queue. This condition violates *The Property of the Return Path*. It now becomes impossible to terminate the token execution of the algorithm without somehow checking every message queue.

5. DISTRIBUTED ALGORITHM SIMULATOR

The Distributed Algorithm Simulator, a C program, is used to study the effect of executing distributed algorithms under the constraint of tokens. The code listing of the Distributed Algorithm Simulator appears in Lloyd (1987). Any distributed algorithm can be implemented on the Distributed Algorithm Simulator by changing the code in one of the code modules. In this thesis, the minimum-weight spanning tree algorithm of Gallager *et al.* (1983) was used to study the effect of tokens.

Below, the typical steps taken by the user in a session with the Distributed Algorithm Simulator are described. In the following section is a presentation of how the Distributed Algorithm Simulator simulates algorithm execution.

5.1. Typical Steps in a Simulation Session

The typical steps taken by the user in a session with the Distributed Algorithm Simulator are specifying the network topology, selecting the algorithm execution strategy, executing the distributed algorithm, and recording results.

5.1.1. Specifying the Distributed System

The Distributed Algorithm Simulator executes a distributed algorithm on a network described by the computational model in Section 2. Recall that a network is modeled by a connected undirected graph in which graph nodes correspond to processors and graph edges correspond to bidirectional communication links.

The user begins by specifying processors. Each processor has associated with it an *Algorithm Start Status* and an *Initial Token Possession Status*.

The Algorithm Start Status specifies whether a processor starts the algorithm. Before algorithm execution begins, all processors are assumed to be in the state *sleep*. Sleeping

processors do not generate or transmit messages. To start algorithm execution, the Distributed Algorithm Simulator wakes up all processors whose Algorithm Start Status is **true**. The awakened processors change their current state and send ordinary messages to neighboring processors according to the algorithm being executed. Processors not initially awakened are awakened upon the receipt of an ordinary message from a neighbor. Eventually, all processors are awakened and participate in the execution of the algorithm.

The Initial Token Possession Status specifies whether a processor starts with a token. In the token execution of the algorithm, a processor may transmit an ordinary message only when it holds a token.

At least one processor must start the algorithm. In the token execution of the algorithm, at least one processor must initially hold a token. Furthermore, at least one of the start processors must also initially hold a token; otherwise, nothing happens. Also, notice that a start processor which does not initially hold a token can not transmit messages until it receives a token message, at which time it would be awakened if it had been sleeping.

After specifying the processors, the user specifies links by selecting the processors at each end of each link.

The algorithm being simulated by the Distributed Algorithm Simulator may use additional information. In the case of the minimum-weight spanning tree algorithm of Gallager *et al.* (1983), link weights must be specified.

The description of the network can be saved in a UNIX file from the Distributed Algorithm Simulator. The user can retrieve a saved network description in a later session.

5.1.2. Selecting the Algorithm Execution Strategy

The user selects the chaotic execution strategy or the token execution strategy. The Simple Token Execution Strategy and its variations can be used alone or can be combined for the token execution strategy. Each Simple Token Execution Strategy variation is explained in detail in Section 4. Briefly the variations are Piggyback Token Messages, Deny Token Start Processor, Leave Return Path, Send Multiple Messages with Each Token, Send Messages on Token Reject, and Multiple Tokens.

5.1.3. Executing the Distributed Algorithm

Three modes of algorithm execution are available: Step Execution, Single Execution, and Auto Execution.

Step Execution simulates one ordinary message delivery. With Step Execution, the user can observe the effect of each ordinary message as it is delivered.

Single Execution completes the currently started algorithm execution. If an execution is not yet started, then a new execution is started by awakening the processors whose Algorithm Start Status is **true**.

Auto Execution simulates two or more algorithm executions in succession, randomizing the network between each execution. Randomizing the network does not change the network topology, processor Algorithm Start Status, or processor Initial Token Possession Status. In the case of the minimum spanning tree algorithm presented in Gallager *et al.* (1983), randomizing the network consists of reassigning link weights according to a random permutation, where each permutation is equally likely.

Section 5.2 provides a detailed description of the execution of distributed algorithms in the Distributed Algorithms Simulator.

5.1.4. Creating Reports

Several reports can be created to record the results of a session with the Distributed Algorithm Simulator. The network topology, algorithm execution strategy, and results of algorithm executions can be saved in a UNIX file. Algorithm execution can be traced by recording all messages sent and delivered in a UNIX file.

5.2. Algorithm Execution on the Distributed Algorithm Simulator

Algorithm execution on the Distributed Algorithm Simulator is accomplished by simulating the transmission and receipt of messages. Figure 1 shows the structure of the Distributed Algorithm Simulator.

The boxes represent data structures; the ellipses represent code modules. Solid arrows represent the flow of ordinary messages; dashed arrows represent the flow of token messages. Dotted lines represent a code module's use of a data structure.

Below a description of data structures and major code modules is followed by a trace of the flow of ordinary messages and token messages.

5.2.1. Data Structures

The data structures used by the Distributed Algorithm Simulator are Network Descriptor, Outgoing Message Queues, and Message Wheel.

Network Descriptor completely describes the network topology. For each processor P, Network Descriptor contains the Algorithm Start Status, Initial Token Possession Status, and current state of P. For each link L, Network Descriptor contains the identifiers of the processors adjacent to L. Also, Network Descriptor contains additional information specific to the algorithm being simulated. For example, the link weights are included in Network Descriptor for the simulation of the minimum-weight spanning tree algorithm of Gallager *et al.* (1983). The user enters the data for Network Descriptor through the Distributed

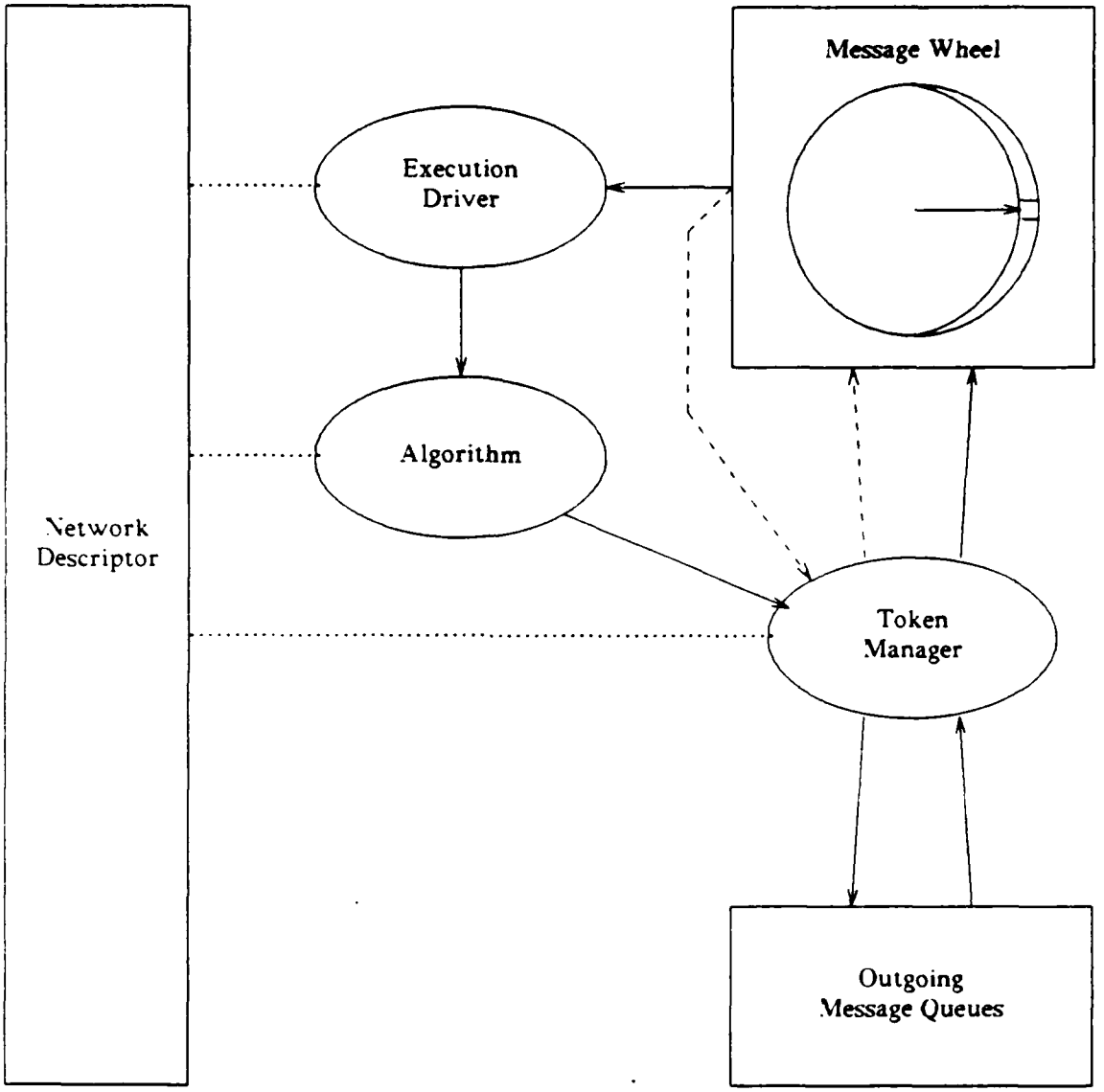


Figure 1. Distributed Algorithm Simulator Structure

Algorithm Simulator prior to algorithm execution.

Outgoing Message Queues contain the ordinary message queue $MSG(P_i)$ for each processor P_i in the network. $MSG(P_i)$ is empty for all P_i at the outset of the the algorithm execution.

Message Wheel stores messages, ordinary and token, until Execution Driver is ready to deliver them. Message Wheel, based on an idea presented in Ulrich (1968), simulates the facilities available to the session layer of the Open Systems Interconnection Model (Tanenbaum, 1981). The session layer assumes messages sent on a link arrive after a finite but unpredictable delay without error and in sequence. Message Wheel delivers messages after a bounded random delay without error and in sequence. Bounded delays allow a fixed sized Message Wheel to be used.

Message Wheel comprises 256 *spokes*, one of which is called the *present time spoke*. Each spoke points to a linked list of messages to be delivered during a particular time unit. The next spoke in the clockwise direction from the present time spoke on Message Wheel points to the messages to be delivered one time unit in the future. After all messages have been delivered for the present time spoke, the next spoke in the clockwise direction becomes the present time spoke.

The delivery time of each message passed to Message Wheel is calculated by first computing a random delay between 1 and t_{max}, where t_{max} is specified by the user. The delay is then added to the delivery time of the previous message on the same link, if such a message exists on Message Wheel. Otherwise, the delay is added to the delivery time of the present time spoke. The message passed to Message Wheel is then added to the spoke corresponding to the calculated delivery time. If the calculated delivery time is 255 or more time units in the future, then Message Wheel wraparound error occurs. Message Wheel wraparound error can be fixed by reducing t_{max} or increasing the number of spokes in Message Wheel. Message Wheel is empty at the outset of the each algorithm execution.

5.2.2. Major Code Modules

The major code modules used by the Distributed Algorithm Simulator are Execution Driver, Algorithm, and Token Manager.

Execution Driver controls the execution of the algorithm. Execution Driver can execute the algorithm in one of three modes: Step Execution, Single Execution, and Auto Execution. In Step Execution, Execution Driver passes one ordinary message from Message Wheel to Algorithm. In Single Execution, Execution Driver keeps passing ordinary messages from Message Wheel to Algorithm until Message Wheel is empty. When Message Wheel is empty, the algorithm execution has terminated. In Auto Execution, Execution Driver runs two or more Single Executions in succession.

Algorithm responds to ordinary messages passed to it according to the distributed algorithm being executed. To respond to an ordinary message with destination P_i, Algorithm may change P_i's state and may send ordinary messages on P_i's adjacent links. Any distributed algorithm can be implemented on the Distributed Algorithm Simulator by changing the code in the Algorithm code module. All other code modules remain unchanged.

Token Manager carries out the currently selected algorithm execution strategy. Suppose processor P_i receives a token message. In general, Token Manager dequeues one or more ordinary messages from the $MSG(P_i)$ in Outgoing Message Queues, passes the ordinary messages to Message Wheel, and passes an appropriate token message to Message Wheel.

Below, the interaction between major code modules and data structures is illustrated by tracing the flow of messages in the Single Execution mode under both the chaotic and token execution strategies.

5.2.3. Message Trace: Chaotic Execution of the Algorithm

In a chaotic execution of the algorithm, a processor sends ordinary message M as soon as it generates M. The sequence of events for the chaotic execution of the algorithm follows.

(1) The user starts the chaotic execution of the algorithm in Single Execution mode from Execution Driver.

(2) Execution Driver passes wakeup messages to Algorithm - one at a time - one for each processor P_i whose Algorithm Start Status is **true**.

(3) In response to each ordinary message passed in, Algorithm may change P_i's current state or generate ordinary messages. Each ordinary message generated by Algorithm is passed to Token Manager.

(4) Since the chaotic algorithm execution strategy is specified, Token Manager passes each ordinary message directly to Message Wheel.

(5) Execution Driver gets the next ordinary message to be delivered from Message Wheel and passes it to Algorithm.

(6) Steps 3, 4, and 5 are repeated until Message Wheel no longer has ordinary messages. The current algorithm execution is assumed terminated.

5.2.4. Message Trace: Token Execution of the Algorithm

In a token execution of the algorithm, a processor may send ordinary messages only when it holds a token. Ordinary messages follow the same sequence of events as in the chaotic execution described above with the exception of step (4), i.e., ordinary message handling by Token Manager. Token Manager now enqueues each ordinary message passed to it by Algorithm in the appropriate $MSG(P_i)$ in Outgoing Message Queues. Ordinary messages are dequeued by Token Manager as follows.

(1) The user starts the token execution of the algorithm in Single Execution mode from Execution Driver.

(2) Execution Driver passes token messages to Algorithm, one for each processor P_i whose Initial Token Possession Status is **true**.

(3) Token Manager dequeues an ordinary message M from $MSG(P_i)$ and passes M to Message Wheel.

(4) Token Manager sends a token message to a processor adjacent to P_i by passing an appropriate token message to Message Wheel.

(5) Execution Driver gets the next message to be delivered from Message Wheel. If the next message is an ordinary message, Execution Driver passes it to Algorithm. If the next message is a token message, Execution Driver passes it to Token Manager. Token Manager responds to the token message according to steps 3 and 4. (In the actual Distributed Algorithm Simulator code implementation, Execution Driver requests the next ordinary message. Any token message to be delivered from Message Wheel before the next ordinary message is passed to Token Manager.)

(6) When Message Wheel no longer has ordinary messages to deliver, the current algorithm execution is assumed terminated.

6. SIMULATION STUDIES

The Simple Token Execution Strategy and its six variations are studied with the minimum-weight spanning tree (MST) algorithm of Gallager *et al.* (1983) (Adams, 1986). The purpose of the studies is to find the most efficient combination of Simple Token Execution variations under different numbers of tokens. Efficiency is measured in terms of number of additional messages and additional algorithm execution time needed by the token execution over the chaotic execution.

Below, descriptions of the network topologies and statistics collected are followed by the simulation results and analysis.

6.1. Network Topologies

Data were collected on the MST algorithm executing on three network topologies: fully interconnected (16 processors), cluster (30 processors), and mesh (25 processors). These topologies are shown in Figure 2 through Figure 4. The mesh network topology has wraparound connections, e.g., processor A is connected to processor U. The three topologies are based on real applications and vary in the number of links per processor and the diameter. The fully interconnected topology has 15 links per processor and diameter 1. The cluster topology has 5 links per processor and diameter 3. The mesh topology has 4 links per processor and diameter 4.

The MST algorithm was executed several times on each network topology using the Distributed Algorithm Simulator. The simulator forced the algorithm to find a different MST each execution by randomly permuting the link weights between successive executions.

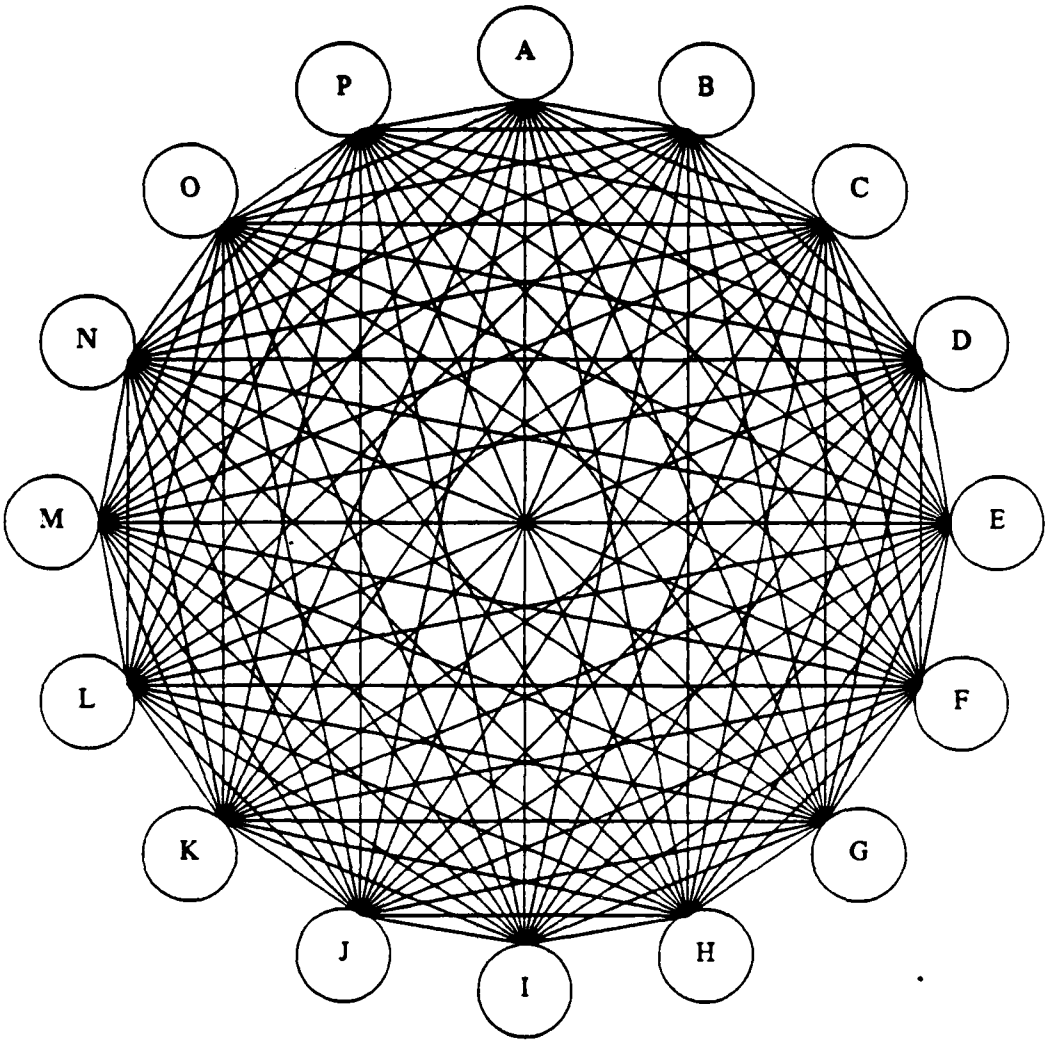


Figure 2. Sixteen-Processor Fully Interconnected Network Topology

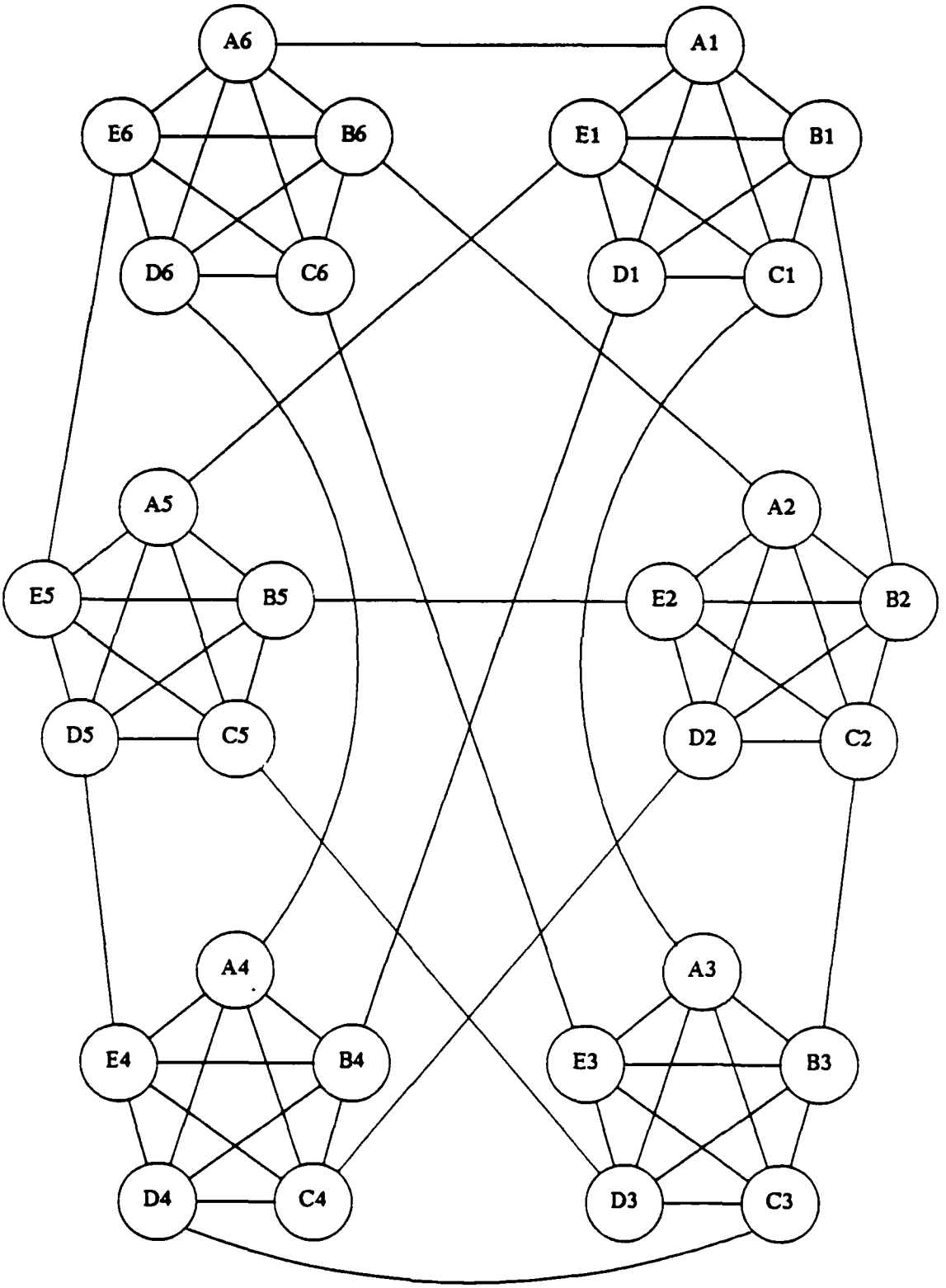


Figure 3. Thirty-Processor Cluster Network Topology

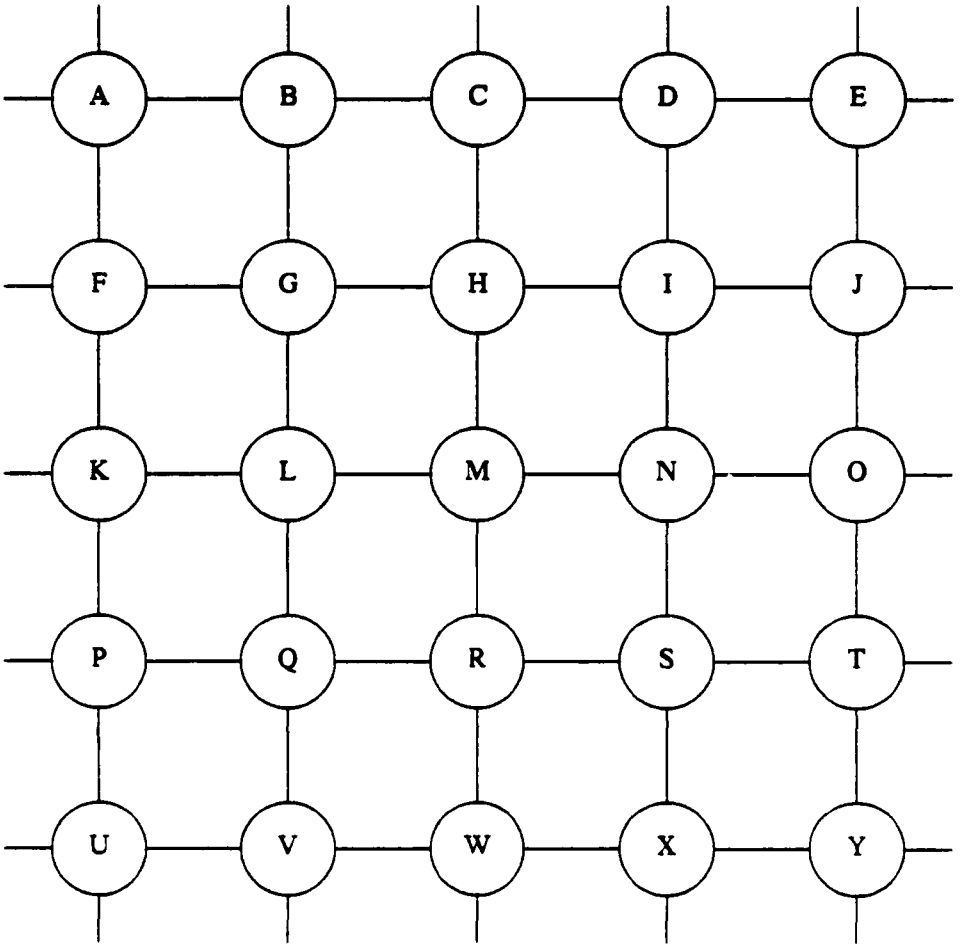


Figure 4. Twenty-Five-Processor Mesh Network Topology

6.2. Statistics Collected

The average number of ordinary messages, average number of token messages, and average execution time were recorded for the Simple Token Execution Strategy and its six variations. A t test (Witte, 1980) for two independent samples was used to determine whether the difference between the data collected for a Simple Token Execution variation and the Simple Token Execution Strategy itself qualifies as a probable or improbable outcome under the null hypothesis. The null hypothesis, the averages over 88 executions are equal, is rejected at the 0.05 level of significance if $t \geq 1.667$. If the null hypothesis is retained, then the difference between the averages is taken as 0.

The t test on the average number of ordinary messages for a given network topology shows the average to be nearly constant over all chaotic and token executions of the MST algorithm. In other words, all message overhead in the token execution is token messages. Therefore, to find the most efficient token execution strategy, we concentrate on minimization of the number of token messages and minimization of the execution time.

6.3. Results and Analysis

The Piggyback Token Execution saves messages and time through combining token messages with ordinary messages whenever possible. Case Study 1 was carried out to determine how much message and execution time savings result from the Piggyback Token Execution. Case Study 2 determined the average token and average execution time performances for combinations of the remaining Simple Token Execution variations in which the number of processors initially holding a token is 1, 2, 4, 8, 16, and N, where N is the total number of processors. Case Study 3 was carried out to compare the performance of the most efficient combination of variations from Case Study 2 for different numbers of processors initially holding tokens with the performance of the chaotic execution.

6.3.1. Case Study 1

The average number of additional token messages and average additional execution time needed by the Simple Token Strategy over the Piggyback Token Execution is presented in Table 1.

The Piggyback Token Execution is clearly more efficient in terms of token messages and execution time. The Piggyback Token Execution is used for all token executions in Case Study 2 and Case Study 3.

Table 1. Performance of Simple Token Execution and Piggyback Token Execution

Token Execution Strategy	Network Topology					
	Full		Cluster		Mesh	
	token messages	time	token messages	time	token messages	time
Simple	500.0	4662.6	683.6	6149.6	508.6	4576.5
Piggybacking	152.0	2745.0	249.6	3744.0	181.9	2769.8
Analysis :						
t	69.416	39.210	68.197	38.813	46.029	25.142
Change	dn 69.9%	dn 41.1%	dn 63.5%	dn 39.1%	dn 64.2%	dn 39.5%

6.3.2. Case Study 2

The Multiple Token Execution is combined with the remaining four Simple Token Execution variations: Deny Token Start Processor, Leave Return Path, Send Multiple Messages with Each Token, and Send Messages on Token Reject. Each of the twelve combinations possible with the four remaining variations is assigned a shortened name in Table 2. The P in the shortened name serves as a reminder that the Piggyback Token Execution is combined with the twelve variation combinations.

The average number of token messages and average execution time for the twelve combinations of token execution variations and the number of processors initially holding a token is 1, 2, 4, 8, 16, and N are presented in Table 3 through Table 8.

Table 2. Shortened Names of Combinations of Simple Token Execution Variations

Simple Token Execution Variation Combination			Shorted Name
No Deny No Leave	No Multiple Messages	No Reject	P
		Reject	PR
	Multiple Messages	No Reject	PM
		Reject	PMR
Deny No Leave	No Multiple Messages	No Reject	PD
		Reject	PDR
	Multiple Messages	No Reject	PDM
		Reject	PDMR
No Deny Leave	No Multiple Messages	No Reject	PL
		Reject	PLR
	Multiple Messages	No Reject	PLM
		Reject	PLMR

Table 3. Performance of Token Execution Variations with 1 Processor Holding a Token

Token Execution Strategy	Network Topology					
	Full		Cluster		Mesh	
	token messages	time	token messages	time	token messages	time
P	152.0	2745.0	249.6	3744.0	181.9	2769.8
PR	111.0	2529.3	196.1	3452.1	160.8	2675.9
PM	144.4	2713.2	234.5	3643.9	170.2	2699.0
PMR	98.5	2449.5	181.8	3407.9	139.7	2541.4
PD	150.1	2704.8	246.5	3646.1	184.6	2741.5
PDR	107.0	2485.6	198.2	3417.5	157.7	2608.1
PDM	144.4	2672.1	231.1	3570.5	173.4	2709.2
PDMR	94.5	2385.3	177.0	*3277.3*	144.8	2549.1
PL	144.0	2689.0	231.2	3634.0	173.0	2726.6
PLR	101.3	2460.4	182.1	3382.0	143.0	2538.8
PLM	134.3	2634.6	224.4	3613.7	162.6	2668.7
PLMR	*87.6*	*2350.1*	*168.6*	3294.9	*131.8*	*2495.0*

Table 4. Performance of Token Execution Variations with 2 Processors Holding a Token

Token Execution Strategy	Network Topology					
	Full		Cluster		Mesh	
	token messages	time	token messages	time	token messages	time
P	185.3	1743.7	253.5	2613.3	190.6	1945.9
PR	117.8	1566.7	206.6	2361.3	162.9	1786.4
PM	177.7	1772.8	241.9	2551.4	182.6	1994.8
PMR	110.3	1547.0	188.1	2367.4	146.0	1776.1
PD	175.4	1660.3	249.5	2561.4	183.4	1881.8
PDR	112.2	1464.7	200.3	2381.8	158.5	1741.9
PDM	173.2	1689.4	230.8	2540.9	168.1	1877.1
PDMR	109.0	1480.7	182.2	2302.6	144.9	1723.5
PL	174.0	1619.6	232.7	2445.5	171.0	1810.5
PLR	109.4	1501.2	182.7	2207.1	147.4	1700.5
PLM	160.7	1619.8	215.6	2345.6	155.1	1732.1
PLMR	*97.8*	*1461.1*	*170.6*	*2166.6*	*132.9*	*1630.2*

Table 5. Performance of Token Execution Variations with 4 Processors Holding a Token

Token Execution Strategy	Network Topology					
	Full		Cluster		Mesh	
	token messages	time	token messages	time	token messages	time
P	216.2	1083.1	260.7	1779.7	200.1	1412.7
PR	150.1	986.8	219.5	1620.5	175.1	1272.6
PM	207.9	1081.1	245.4	1741.5	174.1	1291.2
PMR	140.0	983.6	204.2	1609.8	160.8	1241.4
PD	188.1	951.4	244.2	1640.1	174.3	1193.1
PDR	129.6	*809.6*	208.4	1536.7	164.0	1168.1
PDM	179.5	913.0	227.5	1642.7	157.3	1152.5
PDMR	124.8	8..4.4	189.2	1525.6	147.6	1133.1
PL	188.6	951.3	211.5	1499.3	152.9	1127.3
PLR	126.2	860.0	178.8	1429.7	144.1	1111.0
PLM	173.8	931.6	199.3	1496.6	141.5	1117.9
PLMR	*117.7*	870.8	*168.8*	*1388.2*	*127.6*	*1059.7*

Table 6. Performance of Token Execution Variations with 8 Processors Holding a Token

Token Execution Strategy	Network Topology					
	Full		Cluster		Mesh	
	token messages	time	token messages	time	token messages	time
P	232.6	632.5	271.8	1085.7	205.0	789.1
PR	189.8	576.4	240.5	1029.4	195.5	764.3
PM	226.3	637.9	254.5	1063.0	191.1	776.0
PMR	178.2	569.4	227.8	1010.5	181.4	790.9
PD	164.7	473.2	223.4	934.5	164.0	639.2
PDR	138.7	451.7	205.5	904.0	159.8	615.7
PDM	160.2	474.6	210.5	890.6	151.4	631.1
PDMR	135.0	*442.5*	188.3	865.7	147.9	612.7
PL	160.8	518.2	184.1	899.1	128.3	669.1
PLR	133.0	515.1	165.8	860.7	122.5	638.4
PLM	157.4	502.6	165.1	873.8	109.3	625.7
PLMR	*125.2*	470.7	*148.6*	*847.2*	*106.2*	*602.1*

Table 7. Performance of Token Execution Variations with 16 Processors Holding a Token

Token Execution Strategy	Network Topology					
	Full		Cluster		Mesh	
	token messages	time	token messages	time	token messages	time
P	228.2	336.8	283.2	655.7	229.4	486.0
PR	215.5	336.7	275.8	609.9	223.1	484.1
PM	224.0	332.6	266.7	591.9	204.9	456.3
PMR	202.9	316.5	257.6	597.3	201.9	449.0
PD	107.8	255.2	179.2	479.8	114.0	332.0
PDR	105.5	251.2	174.0	468.1	115.4	343.8
PDM	105.6	*248.6*	169.1	*450.7*	111.9	338.4
PDMR	107.0	251.2	163.0	459.1	111.8	*329.5*
PL	62.0	262.8	126.4	509.4	73.2	356.9
PLR	53.3	257.1	118.4	498.6	70.8	355.1
PLM	57.8	254.9	109.9	497.8	62.4	352.4
PLMR	*53.2*	255.9	*101.7*	484.1	*62.2*	353.9

Table 8. Performance of Token Execution Variations with *N* Processors Holding a Token

Token Execution Strategy	Network Topology					
	Full		Cluster		Mesh	
	token messages	time	token messages	time	token messages	time
P	228.2	336.8	317.9	399.5	234.0	333.4
PR	215.5	336.7	304.4	382.9	240.2	342.3
PM	224.0	332.6	301.6	390.9	221.2	318.7
PMR	202.9	316.5	295.9	379.8	225.4	325.8
PD	107.8	255.2	79.1	318.1	51.3	266.7
PDR	105.5	251.2	78.8	311.8	50.9	271.8
PDM	105.6	*248.6*	80.5	309.7	51.3	266.1
PDMR	107.0	251.2	80.2	311.7	51.4	270.1
PL	62.0	262.8	26.2	324.9	16.1	265.4
PLR	53.3	257.1	25.3	315.4	17.0	266.7
PLM	57.8	254.9	26.7	319.6	16.6	276.8
PLMR	*53.2*	255.9	*24.6*	*308.9*	*15.8*	*262.6*

A few trends can be seen in the data, although it should be noted that the data reflect behavior for only one distributed algorithm and may not hold for all distributed algorithms. The trends are shown for the data of the mesh network topology. The fully interconnected topology and cluster topology show the same trends.

(1) Table 9 and Table 10 show that Deny Token Start Processor and Leave Return Path become significantly more effective as the number of processors initially holding a token increases. Deny Token Start Processor becomes more effective as the number of processors initially holding a token increases because a greater number of tokens are denied, i.e., a token message is not sent to a neighboring processor because the neighbor is known to have initially held a token. Leave Return Path becomes more effective as the number of processors initially holding a token increases because the beginning of

Table 9. Comparison of NO Deny Token Start Processor and Deny Token Start Processor

Number of Tokens	Combination of Token Execution Variations			
	P	PR	PM	PMR
1 Token	0%	0%	0%	0%
2 Tokens	dn 3.8%	0%	dn 8.0%	0%
4 Tokens	dn 12.9%	dn 6.4%	dn 9.6%	dn 8.2%
8 Tokens	dn 20.0%	dn 18.3%	dn 20.8%	dn 18.5%
16 Tokens	dn 50.3%	dn 48.3%	dn 45.4%	dn 44.6%
N Tokens	dn 78.1%	dn 78.8%	dn 76.8%	dn 77.2%

Table 10. Comparison of NO Leave Return Path and Leave Return Path

Number of Tokens	Combination of Token Execution Variations			
	P	PR	PM	PMR
1 Token	dn 4.9%	dn 11.1%	dn 4.4%	dn 5.7%
2 Tokens	dn 10.3%	dn 9.5%	dn 15.1%	dn 8.9%
4 Tokens	dn 23.6%	dn 17.7%	dn 18.7%	dn 20.7%
8 Tokens	dn 37.4%	dn 37.4%	dn 42.8%	dn 41.4%
16 Tokens	dn 68.1%	dn 68.3%	dn 69.6%	dn 69.2%
N Tokens	dn 93.1%	dn 92.9%	dn 92.5%	dn 93.0%

the each token's Return Path tends to move with the token.

(2) Table 11 shows that Leave Return Path becomes more effective than Deny Token Start Processor as the number of processors initially holding a token increases. Leave

Return Path becomes more effective because Deny Token Start Processor must use a fixed number of token messages to find out which neighboring processors initially held a token; the number of token messages needed depends on the number of processors initially holding tokens and the number of links in the network.

(3) Table 12 shows that Send Multiple Messages with Each Token is slightly affected by the number of processors initially holding a token. Send Multiple Messages with Each Token may be more effective with distributed algorithms other than the minimum-weight spanning tree algorithm of Gallager *et al.* (1983).

(4) Table 13 shows that Send Messages on Token Reject becomes less effective as the number of processors initially holding a token increases. To understand why Send Messages on Token Reject becomes less effective, consider processor P_i. As the number of processors initially holding a token increases, fewer ordinary messages tend to stay in $MSG(P_i)$ which lowers the chance of finding an appropriate ordinary message to send when P_i must reject the token.

Table 11. Comparison of Deny Token Start Processor and Leave Return Path

Number of Tokens	Combination of Token Execution Variations			
	P	PR	PM	PMR
1 Token	dn 6.2%	dn 9.3%	dn 6.2%	dn 9.0%
2 Tokens	dn 6.8%	dn 7.0%	dn 7.7%	dn 8.2%
4 Tokens	dn 12.3%	dn 12.1%	dn 10.0%	dn 13.6%
8 Tokens	dn 21.8%	dn 23.3%	dn 27.8%	dn 28.2%
16 Tokens	dn 35.7%	dn 38.7%	dn 44.3%	dn 44.4%
N Tokens	dn 68.6%	dn 66.5%	dn 67.6%	dn 69.3%

Table 12. Comparison of NO Send Multiple Messages with Each Token and Send Multiple Messages with Each Token

Number of Tokens	Combination of Token Execution Variations					
	P	PR	PD	PDR	PL	PLR
1 Token	dn 6.5%	dn 13.1%	dn 6.1%	dn 8.2%	dn 6.0%	dn 7.9%
2 Tokens	dn 4.2%	dn 10.4%	dn 8.3%	dn 8.6%	dn 9.3%	dn 9.8%
4 Tokens	dn 13.0%	dn 8.2%	dn 9.8%	dn 10.0%	dn 7.5%	dn 11.5%
8 Tokens	dn 6.8%	dn 7.2%	dn 7.7%	dn 7.4%	dn 14.8%	dn 13.3%
16 Tokens	dn 10.7%	dn 9.5%	0%	0%	dn 14.9%	dn 12.1%
N Tokens	dn 5.5%	dn 6.2%	0%	0%	0%	0%

Table 13. Comparison of NO Send Messages on Token Reject and Send Messages on Token Reject

Number of Tokens	Combination of Token Execution Variations					
	P	PM	PD	PDM	PL	PLM
1 Token	dn 11.6%	dn 17.9%	dn 14.6%	dn 16.5%	dn 17.4%	dn 19.0%
2 Tokens	dn 14.5%	dn 20.1%	dn 13.5%	dn 13.8%	dn 13.8%	dn 14.3%
4 Tokens	dn 12.5%	dn 7.6%	dn 5.9%	dn 6.2%	dn 5.8%	dn 9.9%
8 Tokens	dn 4.6%	dn 5.1%	0%	0%	dn 4.5%	0%
16 Tokens	0%	0%	0%	0%	0%	0%
N Tokens	0%	0%	0%	0%	0%	0%

6.3.3. Case Study 3

Case Study 3 compares the performance of the *most efficient* combination of the twelve Simple Token Execution variations from Case Study 2 with different numbers of processors initially holding tokens with the performance of the chaotic execution. The results appear in Table 14. (Notice that total messages are given, i.e., ordinary messages + token messages.)

Table 14. Performance Comparison of Chaotic Execution and Most Efficient Combinations of Token Execution

Execution Strategy	Most Efficient Combination	Network Topology					
		Full		Cluster		Mesh	
		total messages	time	total messages	time	total messages	time
Chaotic Execution		349.4	227.8	426.5	306.1	316.2	246.8
1 Token	PLMR	427.0 up 22.2%	2350.1 up 931.7%	600.9 up 40.9%	3294.9 up 976.4%	453.9 up 43.5%	2495.0 up 910.9%
2 Tokens	PLMR	443.2 up 26.8%	1461.1 up 541.4%	600.8 up 40.9%	2166.6 up 607.8%	452.3 up 43.0%	1630.2 up 560.5%
4 Tokens	PLMR	469.0 up 34.2%	870.8 up 282.3%	601.8 up 41.1%	1388.2 up 353.5%	449.9 up 42.3%	1059.7 up 329.4%
8 Tokens	PLMR	469.6 up 34.4%	470.7 up 106.6%	580.2 up 36.0%	847.2 up 176.8%	426.0 up 34.7%	602.1 up 144.0%
16 Tokens	PLMR	404.6 up 15.8%	2[illegible]9 up 12.3%	530.8 up 24.5%	484.1 up 58.2%	389.6 up 23.2%	353.9 up 43.4%
All Tokens	PLMR	404.6 up 15.8%	255.9 up 12.3%	454.3 up 6.5%	308.9 up 0%	337.0 up 6.6%	262.6 up 6.4%

The *most efficient* combination of token strategies is actually most efficient in terms of token messages. The execution time of the most message efficient combination nearly equals the execution time of the most time efficient combination in nearly all cases. All processors are initially awakened in the chaotic execution of the MST algorithm.

The most efficient combination of variations is Piggyback Token Messages, Leave Return Path, Send Multiple Messages with Each Token, Send Messages on Token Reject, and Multiple Tokens with N tokens. This combination uses 6.5% to 15.8% more messages and 0% to 12.3% more execution time than the chaotic execution, depending on the network topology.

7. CONCLUSIONS AND OPEN PROBLEMS

The Simple Token Execution Strategy and its six variations have been presented, studied, and compared with the chaotic execution strategy. Below a summary of the important results is followed by a brief description of two additional Simple Token Execution Strategy variations left as open problems.

7.1. Summary of Results

A few trends can be seen in the data presented in Section 6, although it should be noted that the data reflect behavior for only one distributed algorithm and may not hold for all distributed algorithms.

(1) Deny Token Start Processor and Leave Return Path become significantly more effective as the number of processors initially holding a token increases.

(2) Leave Return Path becomes more effective than Deny Token Start Processor as the number of processor initially holding a token increases.

(3) Send Multiple Messages with Each Token is slightly affected by the number of processors initially holding a token.

(4) Send Messages on Token Reject becomes less effective as the number of processors initially holding a token increases.

7.2. Open Problems

The analyses of the two Simple Token Execution Strategy variations described below are left as open problems.

7.2.1. Owing Queue

For every processor P_i, *Owing Queue* uses a queue OWING(P_i) in addition to MSG(P_i). Every processor P_i is allowed to send all ordinary messages in MSG(P_i) when P_i holds a token, but P_i is required to enqueue the destination link identifier L in OWING(P_i) for each ordinary message sent. More specifically, when P_i receives a token message, P_i dequeues and sends *all* ordinary messages in MSG(P_i), enqueues the destination link in OWING(P_i) for each ordinary message if and only if the destination link is not already in OWING(P_i), dequeues the first destination link L from OWING(P_i), and sends a token message on link L.

7.2.2. Leave Middle of Return Path

Leave Middle of Return Path applies only to a fully interconnected network and requires that each processor know the identifier of the processors at the end of its adjacent links. Consider processor P_b. Assume P_b holds the token, follows processor P_a in the return path, and has one ordinary message in MSG(P_b) with destination P_c. Processor P_b sends the last ordinary message to P_c followed by a token message with a special flag telling P_c to return the token to P_a instead of P_b. Processor P_b has removed itself from the middle of the Return Path.

8. REFERENCES

Adams, R. A. (1986), *Distributed Deadlock Detection for Communicating Processes*, Ph.D. Dissertation, University of Illinois, Urbana-Champaign, Illinois.

Afek, Y. and E. Gafni (1984), "Simple and Efficient Distributed Algorithms for Election in Complete Networks," *Proceedings: Twenty-Second Annual Allerton Conference on Communication, Control, and Computing*, University of Illinois, Urbana-Champaign, Illinois, pp. 689-698.

Awerbuch, B. (1985), "Complexity of Network Synchronization," *Journal of the ACM*, Vol. 32, No. 4, October 1985, pp. 804-823.

Awerbuch, B. (1987), "Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election, and related problems," *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pp. 230-240.

Chandy, K. M. and L. Lamport (1985), "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, Vol. 3, No 1, February 1985, pp. 63-75.

Gallager, R. G., P. A. Humblet, and P. M. Spira (1983), "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 1, January 1983, pp. 66-77.

Liestman, A. L. and Peters, J. G. (1986), *Distributed Algorithms*, Technical Report TR86-10, School of Computing Science, Simon Fraser University, Burnaby B.C., Canada.

Lloyd, M. J. (1987), *Token Execution Strategies for Distributed Algorithms: Simulation Studies*, M.S. Dissertation, University of Illinois, Urbana-Champaign, Illinois.

Peterson, L. L. (1982), "An $O(n\log n)$ Unidirectional Algorithm for the Circular Extrema Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 4, October 1982, pp. 758-762.

Stallings, W. (1985), *Data and Computer Communications*, Macmillan, New York, New York.

Tanenbaum, A. S. (1981), *Computer Networks*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Tiwari, P. and M. C. Loui (1986), "Simulation of Chaotic Algorithms by Token Algorithms," *Distributed Algorithms on Graphs*, edited by E. Gafni and N. Santoro, Carleton University Press, Ottawa, Ontario, Canada, pp. 143-152.

Ulrich, E. G. (1968), "Serial/Parallel Event Scheduling for the Simulation of Large Systems," *Proceedings - 1968 ACM National Conference*.

Witte, R. S. (1980), *Statistics*, Holt, Rinehart, and Winston, New York, New York.

END

10-87

DTIC